Hindawi Publishing Corporation
International Journal of Distributed Sensor Networks
Volume 2013, Article ID 659518, 7 pages
http://dx.doi.org/10.1155/2013/659518

Research Article

Design and Implementation of a State-Driven Operating System for Highly Reconfigurable Sensor Networks

Tae-Hyung Kim

Department of Computer Science and Engineering, Hanyang University, 1271 Sa 3-Dong, Ansansi, Kyunggi-Do 426-791, Republic of Korea

Correspondence should be addressed to Tae-Hyung Kim; thkimth@gmail.com

Received 22 May 2013; Accepted 17 July 2013

Academic Editor: Tai-hoon Kim

Due to the low-cost and low-power requirement in an individual sensor node, the available computing resources turn out to be very limited like small memory footprint and irreplaceable battery power. Sensed data fusion might be needed before being transmitted as a tradeoff between procession and transmission in consideration of saving power consumption. Even worse, the application program needs to be complicated enough to be self-organizing and dynamically reconfigurable because changes in an operating environment continue even after deployment. State-driven operating system platform offers numerous benefits in this challenging situation. It provides a powerful way to accommodate complex reactive systems like diverse wireless sensor network applications. The memory usage can be bounded within a state transition table. The complicated issues like concurrency control and asynchronous event handling capabilities can be easily achieved in a well-defined behavior of state transition diagram. In this paper, we present an efficient and effective design of the state-driven operating system for wireless sensor nodes. We describe that the new platform can operate in an extremely resource constrained situation while providing the desired concurrency, reactivity, and reconfigurability. We also compare the executing results after comparing some benchmark test results with those on TinyOS.

1. Introduction

With the aid of rapid advances in microelectromechanical systems technology, it is of great convenience to construct a smarter world if we deploy a network of wireless sensor nodes to a target field [1]. A wireless sensor node consists of a low-cost and low-power microcontroller with memory, a short-range radio frequency (RF) communication unit, and an integrated unit of multifunctional sensors in general. For example, a typical configuration uses an 8-bit ATmega microcontroller, which has an RF module with 5–50 m communication range and a code memory of 128 kB and a data memory of about 20 kB as a sensor node in the finite state machine based operating system implementation. We then can construct a wireless sensor network (WSN) using largely deployed sensor nodes to a target field as many as we need, all of which should be properly programmed to control sensing units and process sensed data according in order to accommodate various missions defined in a node.

Although the previously described features of WSNs enable us an opportunity of a wide range of smart applications, the computational environment can be rather uniquely or differently characterized by at least three notable aspects. First, we, as operating system architects, should be aware of the resource scarcity in an embedded sensor node. In other words, the computing power of microcontroller is very low, the memory contains just a few hundred kilobytes, and the supplied battery power is limited and even worse mostly irreplaceable. Second, a WSN application is a heavily distributed computing platform consisting of tens of thousands of autonomously cooperative processes. Even in a conventional computing situation, writing a distributed program is not an easy task at all. If we have to write a heavily distributed program in a self-organizing network under lots of resource constraints in terms of memory footprints and computing power, it seems to be almost impossible to write such a correct program. Last but not least, we are not writing an everlasting program because the wireless sensor network is doomed to be disposable eventually and hardly recyclable afterwards.

Thus, we cannot write a meticulous program in advance. It needs to be rewritten from time to time even in the run time. Therefore, we should minimize the amount of program update cost because the network is endangered to be wasted for application program reconfiguration. Note that data transmission activity is the single largest source of power consumption in WSNs. Sensed data transmission conforms with the existential purpose of WSNs while rewritten code transmission does not. It turns out to be merely an unavoidable maintenance cost that should be spared. Unless there is a proper reprogrammability from a system side, the networks would be useless in spite of the remaining battery power, and it would be almost impractical to reconfigure applications and deploy them to each and every one of sensor nodes in the network. The update cost to accomplish the reconfiguration task would be simply too huge to reconfigure the network application. Thus, the state-driven system needs not only to be able to run in a lightweight manner, but also to be able to support reconfigurability in a minimal update cost. Concurrent processing, asynchronous event handling, and data-centricity should be properly supported as well. The finite state machine model as an execution platform can be a viable solution to meet those seemingly contradictory requirements in the design of a new operating system.

In this paper, we explore a state machine based execution model as an ideal operating system design for a networked sensor node and present its implementation with some benchmark tests. In summary, the design goals for the new operating system architecture are as follows. First, the system size should be small enough to reside on the small memory footprint in a sensor node. In recent technology, the flash memory size for this purpose is about 128 kB. Second, the reconfiguration is indispensable, and the update cost should be maintained as less as possible. Third, the system should be able to provide a friendly programming environment that alleviates the programming burdens in sensor network application programs. Last, in spite of all these seemingly contradictory constraints, the final system performance should not be severely sacrificed as compared to existing operating systems for sensor nodes [1–5].

The remainder of the paper is organized as follows. Section 2 describes the related work in the area of operating system concerns for wireless sensor networks. Section 3, presents the finite state machine model as a sensor node operating system. Under the state machine based operating system like ours, application programs are represented by a set of state transition tables. We present such a state transition table as a sensor node application program. Section 4 presents the architectural structure of the state-driven operating system. We also discuss how to implement the fundamental services of the state-driven operating system like concurrency and reactivity event scheduling. Section 5, shows preliminary benchmark test results that compare ours with TinyOS as presented in TinyBench programs [6]. We conclude this paper in Section 6.

2. Related Work

Considering the nature of embedded systems in wireless sensor networks, traditional real-time embedded operating systems like VxWorks [7] can become the first target of a sensor node operating system platform. However, the target hardware requirements for modern embedded real-time operating system easily exceed those of wireless sensor nodes. Although there are minimized operating systems designed for much smaller and deeply embedded systems, they are developed for control-centric applications like in motor controllers or microwave ovens.

As one of the most popular operating systems for sensor nodes, TinyOS [2] was developed first to possess concurrent and asynchronous event handling capabilities and support distributed and data-centric models with efficient modularity. TinyOS provides a very lightweight thread support and efficient network interfaces. The target hardware for the early version of TinyOS is very much memory constrained, for example, with 8 kB of flash as a program memory and 512 bytes of SRAM as the data memory. Although TinyOS can provide all the essential services for sensor nodes, writing an application program under TinyOS is not an easy task due to the intrinsic complexities in embedded programming for largely distributed sensor networks. There are two derived research thrusts from the underlying operating system toward reconfigurability and ease of using the network.

In the original TinyOS, we need to rewrite a program to come up with a revised binary code. Although such an approach provides maximum flexibility for reconfigurability, the resulting update cost for the entire sensor nodes in the wirelessly connected network would be immensely huge. To lessen the reprogramming cost, TinyOS employs the bytecode interpreter called Maté [3] that runs on it. Using a virtual machine like Maté, the reprogramming cost may be confined to the reasonable amount, but we cannot expect the same degree of necessary flexibility in writing an application program. Later, the TinyOS team published an additional work on this matter that is to grant a more customizable and extensible abstraction boundary in the virtual machine [4]. Other research groups developed an operating system called SOS [5] that provides a modular binary update approach to strengthen the reprogrammability with an acceptable update cost.

TinyOS has been developed as a tailored operating system only for microsensor nodes. In TinyOS, each node runs a single runtime image that is statically linked and needs to be distributed to remote nodes via costly wireless communication links. Even a single point of code change requires an entire image uploading from the beginning. When we define update cost as the sum of all intangible efforts required to reconfigure the sensor network, there are two distinct approaches to save it: script language based approach using virtual machine or modular binary upgrades using dynamic linking mechanism. Maté [3] provides virtual machine framework for TinyOS, which is the first attempt at providing a script language description for application scenarios. It is easy to reconfigure without having to redistribute an entire image because of preinstalled virtual machine at each node. However, the script language based description severely restricts the flexibility to update. SOS [5] adopts dynamic linking approach in modular basis to provide dynamic reconfigurability. No flexibility is restricted in such system, but it is also possible to incur relatively higher update cost for simple application-level modifications like parameter tuning. Moreover, the most flexible mechanism that encompasses a whole spectrum of update costs from full image upgrades like in TinyOS to simple parameter tuning like in sensor network management system (SNMS) [8] is proposed as (Dynamic Virtual Machine) DVM [9] that is equipped with SOS-like kernel for dynamic linking and a virtual machine implemented on top of SOS.

All of the above approaches share a common background, which is an effort to come up with tailored operating system architecture for sensor nodes in emphasis of embedded system constraints over general-purpose functionality. They rooted in the same origin that is the de facto standard sensor node operating system, TinyOS. Other related works [10] include synthesizing software programs where a finite state machine (FSM) description language is used to express embedded control applications. Using a state machine to write an embedded control program can be quite straightforward and well written as in Gomez's article [11]. Unfortunately, the authors in [10] assume that the FSM part should be operated on top of a real-time OS that will give them the necessary concurrency. Since wireless sensor network applications are natural candidates for mechanization as a state machine, it would be a meaningful attempt to design and implement state machine based operating system architecture for sensor nodes. In this paper, we present a full-fledged finite state machine execution platform as a sensor node operating system. The earlier version of conceptual design on FSM-based operating system has been presented in [12].

3. FSM-Based Program Execution Platform

Writing a complicated reactive control program is a difficult task. To deal with the complexity in representing the reactive part of the program, Harel [13] introduced a visual formalism called a "statechart." Since then, a state machine has been an effective and powerful model to describe complex reactive and event-driven systems like WSN applications. Our mission is to provide a lightweight execution engine that implements such a well-proven formalism as a programming model. In the finite state machine implementation, a valid input (or event) triggers a state transition, and output is generated accordingly in a way of making progress by moving the current state to the next one as defined in the transition table. Such a state transition takes place instantaneously, and an output function associated with the state transition is invoked. Using this execution mechanism, a finite state machine sequences a series of actions or handles input events differently depending on the state of the machine. Many embedded real-time systems are naturally amenable to the finite state machine model. The model is also suitable to represent the networked sensor applications.

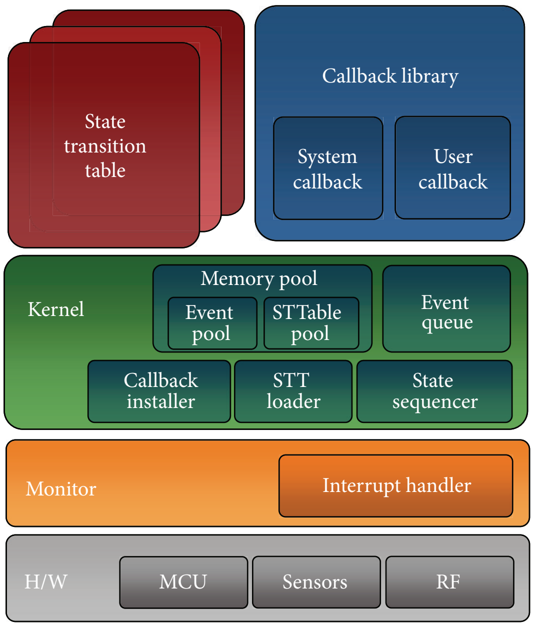


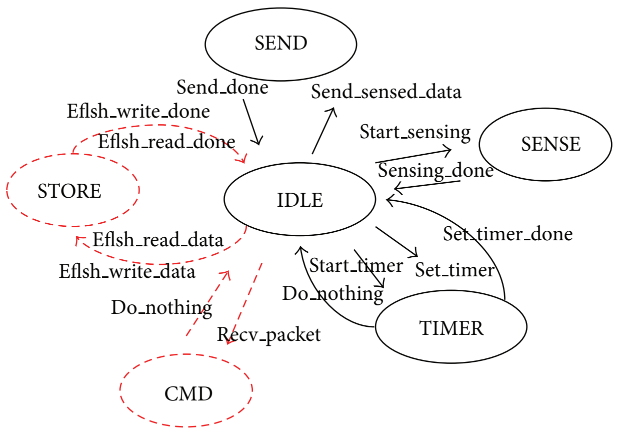
Figure 1: Operating system architecture and its components.

To implement a finite state machine, we need five components as follows: (1) a memory pool consisting of event pool and state transition table, (2) an event queue that stores inputs in a FIFO order, (3) a callback installer with a set of preinstalled callback function library that contains output functions, (4) a state transition table loader with a set of state transition tables to define valid state transitions and their associated callback functions, and finally (5) a state sequencer that accepts an input from an event queue to initiate a proper state transition as shown in Figure 1. Each callback function should satisfy the "run-to-completion" semantics to prevent nested state transitions that are activated by other events captured by the system. Thus, all of those external events are going to be ignored temporarily. They will be stored in the event queue for later services during executing a callback function. Figure 1 schematically depicts the internal architecture of the FSM-driven operating system with its components.

Figure 2 shows a sample application represented by a state modeling diagram. The corresponding state transition table is shown as well, which is executable on FSM. The dotted states and events in the diagram will be represented simply by adding extra state transitions in the table. Update cost becomes merely the size difference between transition tables. The state-driven operating system is suitable to execute such a state transition table with proper callback library functions. This shows why the platform is inherently superior in terms of reconfigurability.

4. Implementation of FSM-Based Sensor Node Operating System

We have implemented the state machine based sensor node operating system on 8-bit MCU ATmega128L (8 MHz)

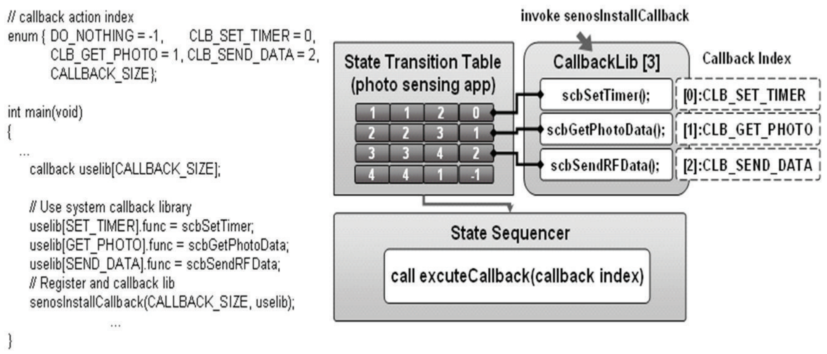


(a)

Current state	Input event	Destination state	Action
IDLE	Start_sensing	SENSE	Select sensor; get sensed data
SENSE	Sensing_done	IDLE	Make packet for sending
IDLE	End_sensed_data	SEND	Send data packet
SEND	Send_done	IDLE	Do nothing
IDLE	Start_timer	TIMER	Start a timer
TIMER	Do_nothing	IDLE	Do nothing
IDLE	Set_timer	TIMER	Set a timer period
TIMER	Set_timer_done	IDLE	Do nothing
IDLE	Eflsh_read_data	STORE	Read sensed data to external flash memory
STORE	Eflsh_read_done	IDLE	Make packet for sending
IDLE	Eflsh_write_data	STORE	Write sensed data to external flash memory
STORE	Eflsh_write_done	IDLE	Do nothing
IDLE	Recv_packet	CMD	Check received packet
CMD	Do_nothing	IDLE	Do nothing

(b)

Figure 2: State modeling and state transition table.

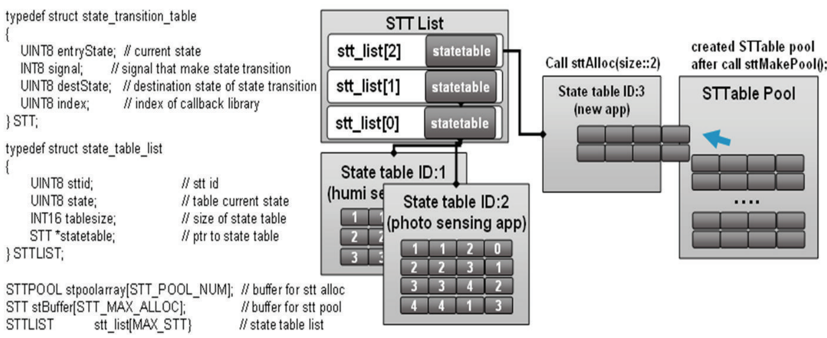


```
// callback action index
enum { DO_NOTHING = -1,      CLB_SET_TIMER = 0,
       CLB_GET_PHOTO = 1, CLB_SEND_DATA = 2,
       CALLBACK_SIZE};

int main(void)
{
  ...
   callback uselib[CALLBACK_SIZE];

   // Use system callback library
   uselib[SET_TIMER].func = scbSetTimer;
   uselib[GET_PHOTO].func = scbGetPhotoData;
   uselib[SEND_DATA].func = scbSendRFData;
   // Register and callback lib
   senosInstallCallback(CALLBACK_SIZE, uselib);
                 ...
}
```

Figure 3: Callback library registry and operation.

equipped with CC2420 (Zigbee) RF module that has a reliable 50 m in-building range. A sensor node has four independent memory banks, each of which has 32 kB flash memory as shown in Figure 2. In the experimental implementation, we hire ten sensor nodes and one sink node (PC). The system is downloaded onto the sensor node that is directly connected to the host PC via a serial communication initially, and then all other nodes obtain the same OS via wireless RF communication. The size of the operating system is 2,686 bytes of code and 292 bytes of data, so less than 3 kB in total. Even if the device drivers are included, the entire size does not exceed 14 kB, which is compact enough to reside in a 32 kB memory bank. The development environment consists of the cross compiler avr-gcc, the fusing program PonyProg2000, and AVR Studio debugging program that is provided by Atmel Corporation. We confirmed the compactness and efficiency of a state machine based operating system which will be presented in Section 5.

The system is divided into a hardware dependent part and an independent one as shown in Figure 1. The hardware dependent component has an event message parser, an interrupt handler, and device drivers for an RF communication



```
typedef struct state_transition_table
{
   UINT8 entryState;  // current state
   INT8 signal;       // signal that make state transition
   UINT8 destState;   // destination state of state transition
   UINT8 index;       // index of callback library
} STT;

typedef struct state_table_list
{
     UINT8 sttid;                // stt id
     UINT8 state;                // table current state
     INT16 tablesize;            // size of state table
     STT *statetable;            // ptr to state table
} STTLIST;

STTPOOL stpoolarray[STT_POOL_NUM];  // buffer for stt alloc
STT stBuffer[STT_MAX_ALLOC];        // buffer for stt pool
STTLIST      stt_list[MAX_STT}      // state table list
```

Figure 4: State transition table structure and management.

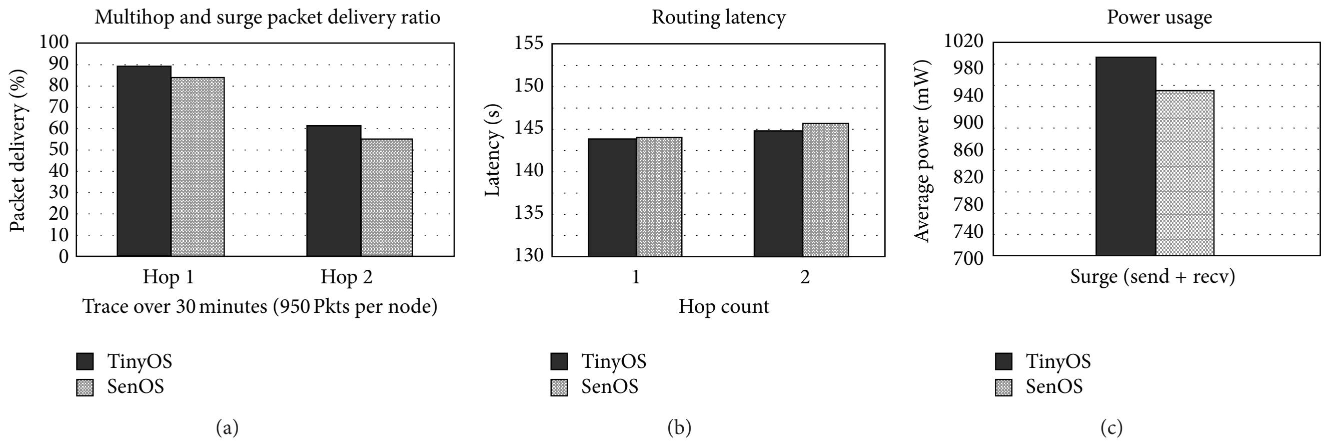


Figure 5: Multihop and surge packet ratio, routing latency, and power usage.

module and a set of sensing units. In the middle level, the kernel architecture is comprised of a state sequencer and an event queue, a state transition table, and a callback library. An input (or event) triggers a state transition if the event queue handles all incoming events in a proper order. The kernel keeps track of the finite state machine in a safe manner by means of employing a *mutex* to guarantee the run-to-completion semantics. The callback library is nothing but a set of predefined functions to help application programmers in advance. The internal structure of the callback library repository and its operation are illustrated in Figure 3.

The operating system architecture also contains a runtime monitor that serves as a dynamic application loader. As mentioned before, the necessity of dynamic reconfiguration is quite demanding. We implement a dynamic application loader that chooses an application to run among a list of legitimate state transition tables. Dynamic reconfiguration can be made in two ways: system and application levels. In system level reconfiguration, some of callback library will be seamlessly replaced by the loader. In application level, it is rather easier than in system level reconfiguration since what we need to be is to update proper rows and/or columns in an existing state transition table. The dynamic loader is in charge of both cases. When the system receives an application reload message via an interrupt from a communication adapter, the monitor puts the kernel into a safe state, stops the kernel, and reloads a new state transition table. Note that the monitor is allowed to interrupt the kernel at any time unless it is in state transition. Since state transition is guarded by a *mutex*, the safety of a state machine is not compromised by such an interruption. Figure 4 shows a brief internal structure of the state transition table and how it can be managed.

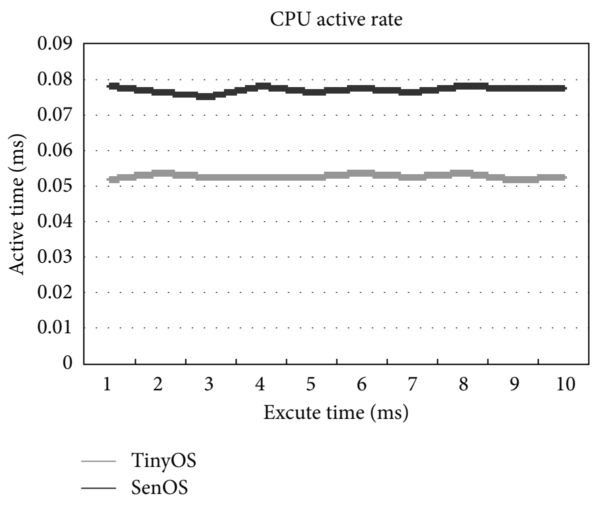
5. Performance Evaluation

The performance data were retrieved to compare them with those of TinyOS through the TinyBench environment [6]. Surge [14] is the simplest test application that was conducted in the TinyBench. It is a direct implementation of a multihop routing algorithm with a couple of sensed data-like temperature and humidity. As shown in Figure 5, the state-driven operating system implementation shows slightly worse performance than those of TinyOS in terms of the packet delivery rate, the routing latency, and the power usage. Note that the difference is still negligible.

To measure the scheduling overhead and the CPUs active rate, we run a blank form which does nothing. For the duration of 500 ms, the number of average instructions and the execution cycles are shown in Figure 6. Still, the overhead and the code size are slightly larger than that of TinyOS, but they are sufficiently affordable in the conventional memory size and CPU performance for a sensor node. Note that

OS version	TinyOS 1.1.0	SenOS 1.0
Hardware platform	Zigbex	
Code size (ROM) Data size (RAM)	1388 bytes 43 bytes	1608 bytes 127 bytes
Benchmark	Scheduling cycle	
Duration	500 ms	
Used instruction set Average instruction cycle	32 465.5 clock	33 474 clock
Benchmark	CPU active time	
Duration	10 ms	
Average usage	51.70 μs (≒ 0.05%)	78.20 μs (≒ 0.07%)

(a)



(b)

Figure 6: Scheduling overhead and CPU active rate.

State-driven OS modules	Code size (bytes)	Execution time (μs)	Average cost (cycles)
senosSTTLoader	68	22.29	92.0
senosInstallCallback	24	8.97	46.5
senosSequencer	107	16.75	152.5
senosExcuteCallback	20	8.12	30.0
eventCreate	29	5.50	34.5
eventGet	7	2.50	12.0
eventFree	24	4.37	34.5
eventMakePool	133	103.00	185.0
seventCreate	74	32.75	107.0
eventGet	16	4.62	40.5
eventFree	41	8.82	54.5
senosMakeSTTPool	55	92.12	73.0
sttAlloc	34	5.67	40.0
sttFree	57	106.50	78.5
senosChangeSTT	50	104.25	80.5

Figure 7: Operating system components: module sizes and system overheads.

the purpose of a finite state machine based sensor node operating system is to improve productivity in application development. We also conducted a larger benchmark like the implementation of the directed diffusion algorithm. Through the test, we confirmed that the development productivity is achieved while the increase of code and data size is restricted within the hardware capability. Based on these benchmark tests, we corroborated the trade-off relationship between productivity and execution performance under finite state machine based operating system. Figure 7 shows a list of component modules for the state-driven operating system implementation and the system overheads in terms of code size and execution time.

6. Conclusions

We have presented the state-driven sensor node operating system as a finite state machine execution platform for wireless sensor nodes. It consists of the kernel, monitor, and replaceable state transition tables and call back libraries. Programmers can easily write a sensor node application and derive executable code using a proper design tool and load the executable code at runtime using the monitor as an agent. Such a state-driven operating system offers a number of benefits. First, its implementation is truly compact and efficient since it is based on a state machine model. Second, it supports dynamic node reconfigurability in a very effective manner using replaceable state transition tables and callback libraries. Third, it can be extended to implement a sensor network management protocol, which is one of the largely untouched regions of sensor network research. Specifically, power management protocol is illustrated to show the dynamic reconfigurability of the system. It gives us an important refinement on application layer support for the sensor network protocol stack. Consequently, programmers can write application programs of higher abstraction for sensor networks. These benefits render the finite state machine platform as a sensor node operating system, which is ideal for most recent WSN applications like intrusion detection [15], energy efficient clustering protocols [16], disaster survivor detection system [17], and so on. We confirmed the compactness and efficiency of the implementation through TinyBench based performance evaluation.

References

[1] J. Yick, B. Mukherjee, and D. Ghosal, "Wireless sensor network survey," *Computer Networks*, vol. 52, no. 12, pp. 2292–2330, 2008.

[2] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," *SIGPLAN Notices*, vol. 35, no. 11, pp. 93–104, 2000.

[3] P. Levis and D. Culler, "Maté: a tiny virtual machine for sensor networks," *ACM SIGPLAN Notices*, vol. 37, no. 10, pp. 85–95, 2002.

[4] P. Levis, D. Gay, and D. Culler, "Active sensor networks," in *Proceedings of International Conference on Networked Systems Design and Implementation*, 2005.

[5] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A dynamic operating system for sensor nodes," in *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services (MobiSys '05)*, pp. 163–176, June 2005.

[6] M. Hempstead, M. Welsh, and D. Brooks, "TinyBench: the case for a standardized benchmark suite for TinyOS based wireless sensor network devices," in *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks (LCN '04)*, pp. 585–586, November 2004.

[7] "VxWorks Platforms Product Overview," http://www.windriver.com/products/vxworks.

[8] G. Tolle and D. Culler, "Design of an application-cooperative management system for wireless sensor networks," in *Proceedings of the 2nd European Workshop onWireless Sensor Networks (EWSN '05)*, pp. 121–132, February 2005.

[9] R. Balani, C.-C. Han, R. K. Rengaswamy, I. Tsigkogiannis, and M. Srivastava, "Multi-level software reconfiguration for sensor networks," in *Proceedings of the 6th ACM and IEEE International Conference on Embedded Software (EMSOFT '06)*, pp. 112–121, October 2006.

[10] M. Chiodo, P. Giusto, A. Jurecska et al., "Synthesis of software programs for embedded control applications," in *Proceedings of the 32nd Design Automation Conference*, pp. 587–592, June 1995.

[11] M. Gomez, "Embedded state machine implementation," in *Embedded System Programming*, pp. 40–50, 2000.

[12] T.-H. Kim and S. Hong, "State machine based operating system architecture for wireless sensor networks," in *Proceedings of the 5th International Conference on Parallel and Distributed Computing: Applications and Technologies (PDCAT '04)*, vol. 3320 of *Lecture Notes in Computer Science*, pp. 803–806, December 2004.

[13] D. Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987.

[14] A. Woo, T. Tong, and D. Culler, "Taming the underlying challenges of reliable multihop routing in sensor networks," in *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems (SenSys '03)*, pp. 14–27, November 2003.

[15] S. Singh, M. Singh, and D. Singh, "Intrusion detection based security solution for cluster based wireless sensor networks," *International Journal of Advanced Science and Technology*, vol. 30, pp. 83–95, 2011.

[16] R. Sheikhpour, S. Jabbehdari, and A. Khadem-Zadeh, "Comparison of energy efficient clustering protocols in heterogeneous wireless sensor networks," *International Journal of Advanced Science and Technology*, vol. 36, pp. 27–40, 2011.

[17] N. Ahmad, N. Riaz, and M. Hussein, "Ad hoc wireless sensor network architecture for disaster survivor detection," *International Journal of Advanced Science and Technology*, vol. 34, pp. 9–16, 2011.